Available at
www.ElsevierMathematics.com
POWERED BY SCIENCE DIRECT®

Discrete Applied Mathematics 138 (2004) 229–251

DISCRETE APPLIED MATHEMATICS

www.elsevier.com/locate/dam

BDDs—design, analysis, complexity, and applications

Ingo Wegener[1]

Fachbereich Informatik LS 2, University Dortmund, 44221 Dortmund, Germany

Received 4 October 2000; received in revised form 13 March 2002; accepted 28 February 2003

Abstract

BDDs (binary decision diagrams) and their variants are the most frequently used representation types or data structures for boolean functions. Research on BDD variants has turned out to be one of the areas where the symbiosis between theoretical investigations in algorithm design and analysis, complexity theory, and applications has led to progress in theory *and* in applications. Here the different roots of the interest in BDDs are described, the main BDD variants and their algorithmic properties are presented, the representation size of selected functions is investigated, lower bound techniques are discussed and applications to algorithmic graph problems and hardware verification problems are presented.

1. Introduction

BPs (branching programs) and BDDs (binary decision diagrams) are synonyms for the same representation type for boolean functions. The notion BPs has been used in complexity theory while the notion BDDs has been used in applications like hardware verification, model checking, or hardware design. We present the definitions in Section 2 and the different motivations for the investigation of BPs and BDDs in Sections 3 and 4, respectively. In the following we only use the notion of BDDs.

The purpose of this survey paper is to show that the cooperation between theoreticians working in complexity theory and on efficient algorithms and practitioners working in areas like design automation has led to new results in theory *and* in applications. New ideas and BDD variants from applications have been investigated with theoretical

[1] Supported in part by DFG We-1066/9.
E-mail address: wegener@ls2.cs.uni-dortmund.de (I. Wegener).

0166-218X/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0166-218X(03)00297-X

tools. The results include the complexity theoretical classification of fundamental algorithmic problems, the design and analysis of algorithms, and upper and lower bounds on the representation size of important functions with respect to the various BDD models. These results had influence on the complexity theory on BDDs. However, it is remarkable that these theoretical results for problems motivated by real-world problems had influence on the way the real-world problems are attacked nowadays. This is one of the few examples of a fruitful cooperation between theoreticians and practitioners.

In Section 5, we introduce the most frequently used BDD variant namely OBDDs (ordered BDDs) and π-OBDDs with a fixed variable ordering π. Efficient OBDD-based algorithms for the important operations on boolean functions are presented. This leads to a fundamental but complexity theoretically hard problem which is called the variable-ordering problem and which is discussed in Section 6. It is the problem how to find an appropriate variable ordering π.

It will turn out that OBDDs have many nice properties—as long as the considered functions can be represented by OBDDs that are not too large. OBDDs are strongly restricted BDDs. Hence, one looks for less restricted BDDs which are good compromises. They should share most of the good algorithmic properties with OBDDs and they should allow that more (and, in particular, more important) functions can be represented in polynomial size (in reasonable size for reasonable input length). Three of these models between OBDDs and general BDDs are discussed in Section 7.

After having concentrated on the complexity theoretical and algorithmic issues of the operations on boolean functions we investigate in Section 8 the representation size of selected functions with respect to the considered models. Among these functions are fundamental arithmetic and storage access functions and some functions which have turned out to be good "theoretical benchmarks." We introduce the central lower bound techniques.

We finish the paper with two case studies of applications. The first one is concerned with algorithmic graph problems, in particular, the maximum flow problem. These are applications pointing into the future. The usual algorithms for graph problems work on graphs represented by adjacency lists fitting into the main memory. Many new problems arise if the external memory has to be used. In these situations the more succinct implicit graph representation by BDDs representing the boolean function describing the edge relation on the vertex set is useful—sometimes. In future, graph problems will become important, where the graphs only can be described implicitly. After this view into the future in Section 9 we discuss in Section 10 an approach how the Pentium bug could have been avoided using OBDDs.

We hope that the reader will find out that BDDs are an interesting or even thrilling subject. In this case, she or he may read the comprehensive presentation of all aspects of BPs and BDDs in Wegener [39].

2. BDDs—a representation of boolean functions

The notion of a decision diagram is so natural that one cannot date the first use of this type of representation. Classification systems used by the famous botanist Carl

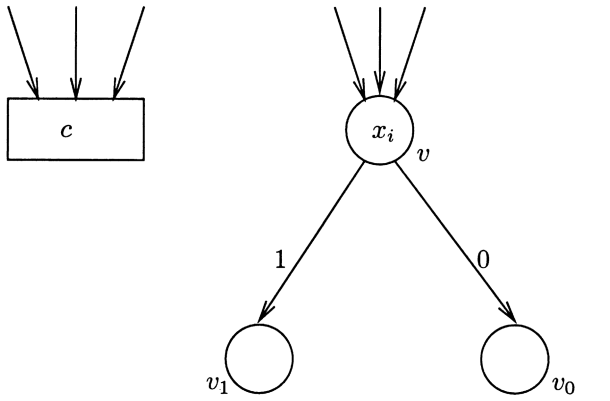


Fig. 1. The components of BDDs.

von Linné in the 18th century can be understood as decision diagrams. The first one to use a decision diagram as representation or data structure for boolean functions was perhaps Lee [24]. In the following we describe BDDs as data structures for boolean functions.

BDDs consist of only two simple types of modules or components.

A *sink* or *output* node is a node without any outgoing edge. Such a node is labeled by a boolean constant $c \in \{0,1\}$. Having reached a sink one outputs the label of this sink.

An *inner* node, *decision* node, or *branching* node is a node v with two outgoing edges one labeled by 1 and the other by 0 and with a label from the variable set $X = \{x_1, \ldots, x_n\}$. If the label is x_i and the outgoing edges reach v_1 and v_0, resp., one chooses the edge whose label equals the value of x_i:

if $x_i = 1$ then go to v_1 else go to v_0.

This statement is also called ITE (if-then-else) (see Fig. 1).

A BDD is a directed acyclic graph consisting of inner nodes and sinks. The number of incoming edges of a node is not restricted. The *size* of a BDD is defined as the number of nodes of the underlying graph. Each node v represents a boolean function $f_v : \{0,1\}^n \to \{0,1\}$ if the underlying variable set X contains n variables. In order to evaluate $f_v(a)$, $a \in \{0,1\}^n$, one starts at v and follows the instructions at the nodes reached on the so-called *computation path* for f_v and a.

In Fig. 2, the function f_v has the value 1 for the input $a = (1,0,0,1)$ as can be seen by following the computation path (- - ->). The path indicated by dotted edges ($\cdots >$) is a graph theoretical path but not the computation path for any input. The reason is that one x_3-node is left via the 1-edge while another x_3-node is left via the 0-edge. Such paths are called *inconsistent* and cause many problems for algorithms working on BDDs. The BDD size BDD(f) of a boolean function $f : \{0,1\}^n \to \{0,1\}^m$ is the

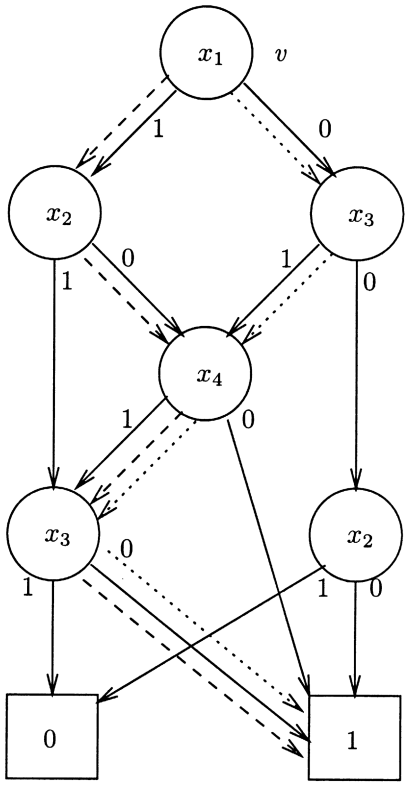


Fig. 2. A BDD with a computation path (- - - >) and an inconsistent path ($\cdots >$).

minimal size of a BDD representing each component f_i of $f=(f_1,\ldots,f_m)$ at some node v_i. Hence, we investigate the *shared* representation of more than one function.

3. Motivations from complexity theory

Already Cobham [15] has shown that the logarithm of the branching program size ($\log \mathrm{BDD}(f_n)$) is asymptotically equal to the space complexity $s(n)$ of nonuniform Turing machines computing f_n. This holds for sequences of boolean functions $f=(f_n)$, $f_n : \{0,1\}^n \to \{0,1\}$, as long as $\mathrm{BDD}(f_n)=\Omega(n)$ and $s(n)=\Omega(\log n)$. The proof is quite simple. The nonuniform Turing machine can simulate a BDD by getting an encoding of the BDD as nonuniform advice. In order to simulate a Turing machine, a BDD can use nodes for all possible configurations. Accepting and rejecting configurations are simulated by 1-sinks and 0-sinks, resp. Nodes simulating nonstopping configurations are labelled by the variable x_i read on the input tape and the outgoing edges point to the configuration reached in the next step for the corresponding value of x_i.

Hence, BDDs are a tool to prove lower bounds on the nonuniform space complexity, in particular, to prove that functions are not contained in LOG-SPACE, or to separate deterministic and nondeterministic complexity classes based on space. However, the best known lower bound for explicitly defined functions is still due to Nechiporuk [27]. It is an $\Omega(n^2/\log^2 n)$ bound for functions with "many subfunctions". The indirect storage access function ISA (see Section 8) is an example of a simple function leading to an $\Omega(n^2/\log^2 n)$ bound by Nechiporuk's method.

This inability to prove lower bounds for the general model has led to the investigation of more restricted models. Some of these restricted models are of purely complexity theoretical interest, but some of them have turned out to have applications. In this paper we discuss some of the models with applications in more detail. The latest results on lower-bound records can be mentioned only. Beame et al. [3] were able to prove exponential lower bounds for BDDs where the length of all computation paths is bounded by $(1+\varepsilon)n$ for some tiny $\varepsilon > 0$. Afterwards, Ajtai [1] presented an exponential lower bound for all BDDs of linear length. This has been generalized to randomized linear-length BDDs by Beame et al. [4].

4. Motivations from algorithmic problems in applications

In complexity theory, BDDs are static representations of boolean functions, while in applications, one manipulates boolean functions and needs dynamic representations which usually are called data structures. In order to develop an appropriate data structure, one has to fix the set of operations that have to be supported. For this purpose, we start with the classical example of BDD applications namely *circuit verification*. The scenario is described in Fig. 3.

We have a specification S, for simplicity, a verified circuit realizing f and a new realization R which is claimed to realize f. Both circuits are given by gate lists $S_1,\ldots,S_k$ and $R_1,\ldots,R_m$, resp. The verification task is to verify that $R \equiv S$, i.e., it has to be proved that the input–output behaviour of R is equal to the input–output behaviour of S. Remember that R and S are typically different approaches to solve the same problem. It is well known and easy to prove that the circuit verification problem is coNP-complete. The BDD approach is to transform $S_1,\ldots,S_k$ and $R_1,\ldots,R_m$ into BDD realizations by simulating the circuits gate by gate. This leads to BDD representations

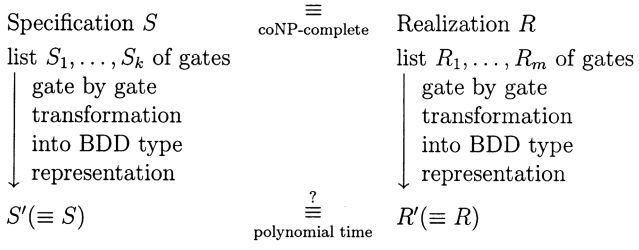


Fig. 3. The classical BDD-based approach for circuit verification.

S' and R', resp., where it is known that $S \equiv S'$ and $R \equiv R'$. Hence, the problem $S \equiv R$ is equivalent to the problem $S' \equiv R'$. We are looking for BDD variants where the equality test can be solved in polynomial time.

Where's the catch? We do not believe that coNP = P and it is not our aim to prove coNP = P by a polynomial-time circuit verification algorithm. One step of the gate by gate transformation may run in polynomial time, i.e., polynomial time with respect to the input size. The resulting representation can be larger than the given one and this may lead to an exponential size blow-up if we carry out a sequence of these steps. Hence, the whole approach is a heuristic one while the single operations are algorithmic problems allowing polynomial-time algorithms. In the following we use this example to motivate the list of important operations. The evaluation problem (given a representation for f and an input a, compute $f(a)$) is easy for all BDD models and will, therefore, not be mentioned in the rest of the paper.

During the gate by gate transformation we consider a gate S_l realizing a boolean operation $\otimes$ on two inputs. We already have BDD representations of the predecessor gates S_i and S_j, where $i, j < l$. The aim is to obtain a BDD representation of the functions represented at S_l from BDD representations of the function represented at S_i and S_j. This operation is called synthesis. It plays a fundamental role, since it has to be solved for all gates. It is essential to control the size of the representations obtained by synthesis steps. BDDs are not unique representations. So it may happen that we obtain an exponential-size representation of a function that also allows a short representation. The minimization problem is to obtain from a representation of f a minimum-size representation with respect to some BDD variant. If each function has a unique minimal-size representation, the minimization process is also called reduction. Our example shows that we finally have to perform an equality test checking whether two BDD nodes represent the same function. Another essential operation is replacement by constants where we have to construct a representation of the subfunction $f_{|x_i=c}$ from a representation of f. We summarize the operations using the following notations: $\wedge$ equals logical AND, $+$ logical OR, $\otimes$ represents an arbitrary boolean operation, $f_{|x_i=c}(a) := f(a_1, \ldots, a_{i-1}, c, a_{i+1}, \ldots, a_n)$, and f denotes a representation of f with respect to the considered BDD variant. The following operations are the essential ones:

- synthesis: $f, g, \otimes \to f \otimes g$,
- minimization: $f \to f$ with minimal size,
- reduction: the case where the result of minimization is unique,
- equality test: $f, g \to$ yes iff $f = g$,
- replacement by constants: $f, x_i, c \to f_{|x_i=c}$.

The following operations are of further interest:

- satisfiability test : $f \to$ yes iff $f \not\equiv 0$,
- replacement by functions : $f, g, x_i \to f_{|x_i=g}$, defined by $f_{|x_i=g}(a) := f(a_1, \ldots, a_{i-1}, g(a), a_{i+1}, \ldots, a_n)$,
- existential quantification : $f, x_i \to f_{|x_i=0} + f_{|x_i=1}$,
- universal quantification : $f, x_i \to f_{|x_i=0} \wedge f_{|x_i=1}$.

The satisfiability test can be realized by negating the result of an equality test of f and the constant 0. Moreover, we can perform an equality test by a nonsatisfiability test for the result of an EXOR-synthesis of f and g. The quantification operations are defined as replacements by constants followed by a synthesis step. The usefulness of the operation quantification will turn out in Section 10. Also replacement by functions can be realized by replacements by constants and synthesis steps, since

$$f_{|x_i=g} = (g \wedge f_{|x_i=1}) + (\bar{g} \wedge f_{|x_i=0}).$$

Replacement by functions is useful if the results of some subcircuits are first replaced with auxiliary variables which later have to be replaced with the functions realized by the subcircuits.

Now it is easy to see the limitations of general BDDs. They support operations like synthesis and replacement by constants. However, the equality test is coNP-complete and the satisfiability test is NP-complete. This holds even if each graph theoretical path contains at most two x_i-nodes for each x_i (see Bollig et al. [6]). Hence, we have to look for restricted BDD variants allowing efficient algorithms for (almost) all the considered operations.

5. OBDDs and π-OBDDs

As already mentioned, OBDDs are the most frequently used data structure for boolean functions, see Somenzi [36] for an OBDD package supporting many features, in particular, solutions for the variable-ordering problem (see Section 6).

A BDD is called *read-once* or *free* (FBDD) if each graph theoretical path contains at most one x_i-node for each variable x_i.

A BDD is called *oblivious* with respect to $s = (s_1, \ldots, s_l)$, $s_i \in X$, if the set of inner nodes can be partitioned into l levels such that level i only contains s_i-nodes and each edge from level i leads to a node on some level $j > i$ or to a sink.

An OBDD (ordered BDD) is an oblivious free BDD.

FBDDs are the most general BDDs without inconsistent paths. Hence, the satisfiability test for f_v can be solved by a DFS (depth first search) traversal checking whether there is a directed path from v to a 1-sink. The sequence s belonging to an OBDD on $X = \{x_1, \ldots, x_n\}$ has a length which can be bounded by n and which contains each variable at most once. We always can add the missing variables and obtain the sequence $s_\pi = (x_{\pi(1)}, \ldots, x_{\pi(n)})$ for a permutation π on $\{1, \ldots, n\}$. Then π is called *variable ordering*.

A π-OBDD is an OBDD respecting the variable ordering π, i.e., it is s_π-oblivious. It is easy to see that π-OBDDs for a fixed variable ordering π are closely related to DFAs (deterministic finite automata). Let G be a π-OBDD for $f : \{0,1\}^n \to \{0,1\}$ where w.l.o.g. $s_\pi = (x_1, \ldots, x_n)$. If we insert dummy nodes labeled by $x_{i+1}, \ldots, x_{j-1}$ (a sink is considered as x_{n+1}-node) on an edge from an x_i-node to an x_j-node where $j > i + 1$, if we add 0- and 1-edges from the 1-sink to the 0-sink and 0- and 1-loops at the 0-sink, then we obtain a DFA for the language $f^{-1}(1)$.

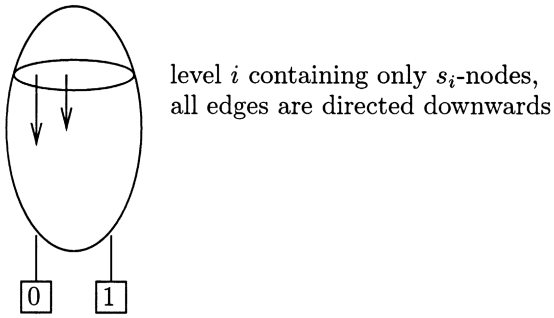


Fig. 4. The illustration of an oblivious BDD.

Indeed, the labelling of the inner nodes of OBDDs is only necessary, since we may skip some levels (the edges in Fig. 4 have different length). This is only possible in acyclic DFAs with the additional property that each node is only reached for the input bit from a fixed position. This freedom may lead to a saving of a size factor of at most $n+1$. This saving is possible for the constant functions defined on n (dummy) variables. One may ask whether the saving of an $\Theta(n)$-factor is also possible for functions essentially depending on n variables. This question has been answered affirmatively by Bollig and Wegener [9].

The fundamental algorithms on π-OBDDs have been presented by Bryant [12] in his paper introducing OBDDs. For the sake of completeness we give a short and informal description of these algorithms. This is necessary to understand the generalized algorithms in Section 7.

The main idea for the synthesis is the same as for DFAs. We perform DFS traversals of the π-OBDDs representing G_f and G_g, resp., where we always choose the 1-successor before the 0-successor. We assume that the variable ordering is given by $(x_1,\ldots,x_n)$. These two DFS traversals are now done "simultaneously" starting at the pair (s_f,s_g) of sinks. Whenever, we consider a pair (v,w) of nodes v and w both labelled by the same variable, we choose the pair of c-successors (v_c,w_c). If we reach the pair (v,w) where v is an x_i-node, w an x_j-node, and $i<j$, then we have skipped in G_g the x_i-level. Then we "wait" in G_g and consider the pair (v_c,w). The case $i>j$ is handled similarly. In the following we denote the size of G_f by $|G_f|$. The new π-OBDD is defined on the product $V_f\times V_g$ and, therefore,

$$|G_{f\otimes g}|\leqslant|G_f|\cdot|G_g|.$$

We have to be careful, since we can skip tests. Because of the fixed variable ordering and the node labels we know "where to wait." Fig. 5 describes how we connect the nodes of $V_f\times V_g$.

The case that v is an inner node and w is a sink (or vice versa) is treated like the case of two inner nodes where the label of v "survives." Moreover, the successors of a sink are the sink again (we introduce dummy loops at the sinks). If f is represented at v^* and g at w^*, then $f\otimes g$ is represented at (v^*,w^*). We simulate the computation paths for f and for g where we sometimes wait in one of the π-OBDDs. Instead

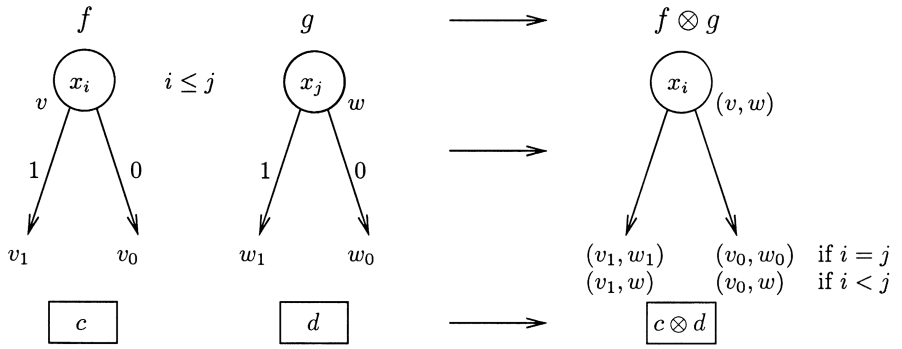


Fig. 5. The synthesis of π-OBDDs for $\pi = id$.

of sinks with the labels $f(a)$ and $g(a)$, resp., we now reach a sink with the label $f(a) \otimes g(a) = (f \otimes g)(a)$.

The naive algorithm will always take time and space $\Theta(|G_f| \cdot |G_g|)$. The first improvement is to create the new π-OBDD by simultaneous DFS-traversals of G_f and G_g respecting the labelling of the edges. This avoids the creation of unreachable nodes.

It is essential that π-OBDDs allow a reduction to a unique minimal-size representation. This again is similar to DFAs. After the elimination of unreachable nodes (which we never create), it is sufficient to merge equivalent nodes. This is for π-OBDDs easier than for DFAs, since π-OBDDs are acyclic and a bottom-up approach is possible. Obviously, it is sufficient to have one sink of each type. However, for π-OBDDs we have to take into account the possibility of skipping tests. Nevertheless, it is sufficient to apply two local reduction rules until they are no longer applicable in order to obtain the reduced π-OBDD, see Fig. 6.

It is obvious that the reduction rules do not change the functions which are represented. Bryant's $O(|G| \log |G|)$ reduction algorithm has been improved by an $O(|G|)$ algorithm due to Sieling and Wegener [33]. However, storage is an important issue if one has to perform the synthesis of π-OBDDs of large size. Hence, it is essential to integrate the reduction process into the synthesis process. Such an integration is possible. Then one works with two hash tables, one for the DFS traversal and the other one for the minimization:

- The *unique-table* contains the nodes of the resulting π-OBDD (the application of the elimination rule is obvious, this table supports the application of the merging rule, no equivalent nodes are inserted by checking whether a node with the same label, the same 1-successor, and the same 0-successor exists already).
- The *computed-table* contains the node pairs of the two given π-OBDDs which have been visited during the simultaneous DFS-traversal (this table prevents the repeated investigation of the same node pairs).

The size of the unique-table is bounded by the size of the resulting reduced π-OBDD while the computed-table may contain $\Theta(|G_f| \cdot |G_g|)$ entries even if the resulting reduced

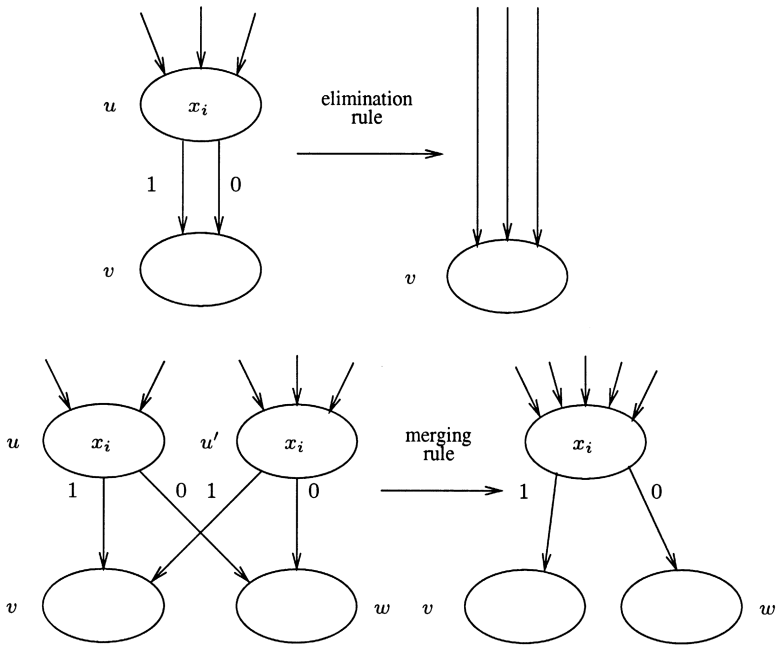


Fig. 6. The reduction rules for π-OBDDs.

π-OBDD consists of a single node. In applications, a heuristic approach is used. The size of the computed-table is limited. Each hash position can take $c = O(1)$ entries. If the $(c+1)$st entry has to be stored, the earliest entry is removed. Note that this cannot lead to wrong results. It is only possible that some parts of the traversal are repeated. Hence, the algorithm may require exponential time in the worst case. Usually, some node pairs are considered more than once. However, there is a good chance that their children are still stored in the computed-table.

The equality test can be performed by the test whether the considered reduced π-OBDDs are isomorphic. Using shared representations (the OBDDs may share nodes if possible), the check whether two functions are equal is equivalent to the test whether they are represented at the same node. Also replacement by constants is easy (see Fig. 7). Since we set $x_i = c$, we can skip x_i-tests and can directly go to the c-successor of the x_i-node. The x_i-nodes can be found during a DFS-traversal. Using ideas from the synthesis process where the reduction process has been integrated we obtain a procedure for replacement by constants with integrated reduction.

6. The variable-ordering problem

The success of OBDDs relies on the freedom to choose a suitable variable ordering. Such a freedom is not given for DFAs for languages containing words of arbitrary length. Many functions with polynomial OBDD size have exponential size for most of

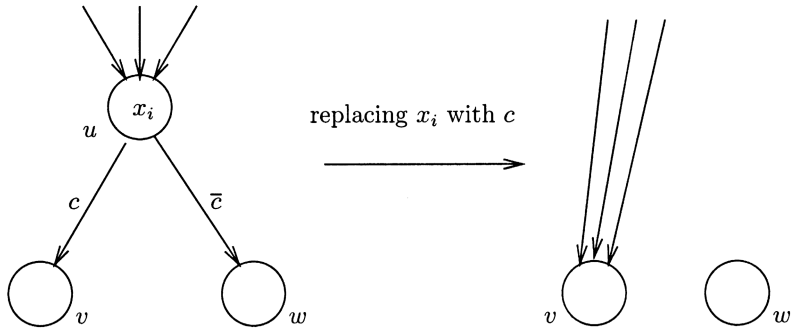


Fig. 7. Replacement by constants.

the variable orderings (see [39]). We only present as an example the function

$$f_n(x_1,\ldots,x_n,y_1,\ldots,y_n) = 1 \leftrightarrow x_i = y_i \quad \text{for all } i$$

testing whether two vectors are equal. This function needs more than 2^n OBDD nodes for the variable ordering $(x_1,\ldots,x_n,y_1,\ldots,y_n)$. The reason is that an OBDD has to represent all subfunctions obtained by replacing the first m variables according to the variable ordering. All different assignments of the x-variables lead to different subfunctions. The OBDD size equals $3n+2$ for the variable ordering $(x_1,y_1,x_2,y_2,\ldots,x_n,y_n)$. It is also not too difficult to prove an exponential lower bound for all but an exponentially small fraction of all variable orderings.

The search for an optimal variable ordering for a function given by an OBDD is an NP-hard problem [7] which even is hard to approximate [32]. Hence, we have to live with heuristic algorithms. The gate by gate transformation of a circuit is started with an arbitrary variable ordering (or a variable ordering computed heuristically from the circuit representation). Whenever the OBDD gets too large, one looks for a better variable ordering. This approach takes into account the experience that certain parts of a circuit have a small representation for variable orderings which are bad for other parts of the circuit. The representation of gates whose successors have been considered can be deleted. However, the synthesis of a function with linear π-OBDD size and a function with linear π'-OBDD size can result in a function with exponential OBDD size (for any variable ordering).

The best results for heuristic reordering of an OBDD have been obtained with evolutionary algorithms [16] and with simulated annealing [5]. However, these approaches are too time consuming to be applied again and again during a gate by gate transformation of a circuit into an OBDD. Despite several sophisticated improvements the fastest and often quite successful heuristic is the sifting algorithm due to Rudell [30]. It chooses a variable and looks for the best position of this variable if the relative ordering of all other variables remains fixed. For this purpose the swap operation is essential. It exchanges the ordering of two neighboured variables. This is a local operation, since it only affects the two neighboured levels of the OBDD labelled with these variables. Also a swap needs an integrated reduction for the considered levels.

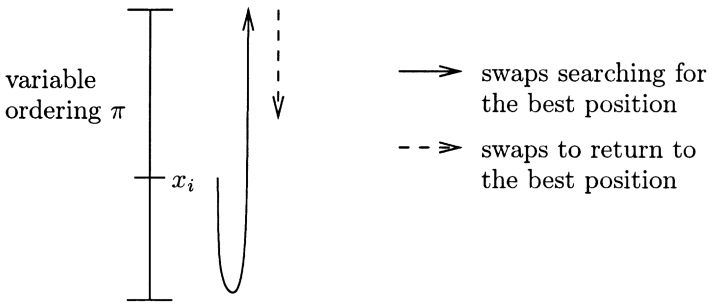


Fig. 8. A sifting step for x_i.

The variable x_i is first swapped to the nearer end of the ordering and then swapped to the other end and finally to the best position (see Fig. 8).

The sifting process is aborted if the increase of the OBDD size gets too large. It has been observed in experiments that the sifting process is almost always only aborted during the sift-down phase and not during the sift-up phase. Theoretical results by Bollig et al. [5] have explained this effect. The OBDD size can be at most doubled during the sift-up phase while sift-down can lead to a quadratic size increase.

Hence, the essential operation in an OBDD package is synthesis with reduction followed by a heuristic reordering. From a theoretical point of view one may ask whether such a step with optimal reordering may lead to a multiplicative size increase, i.e., are there functions depending essentially on $\Omega(n)$ of the considered n variables such that

$$\text{OBDD}(f \otimes g) = \Omega(\pi\text{-OBDD}(f) \cdot \pi\text{-OBDD}(g)).$$

For $f \otimes g$ we consider an optimal variable ordering by investigating the OBDD instead of the π-OBDD complexity. Bollig and Wegener [9] have presented functions f and g with such a size increase. Let f be the direct storage access function $\text{DSA}_n(a,x)$ (also called multiplexer) defined for the address vector $a = (a_{k-1}, \ldots, a_0)$ describing as binary representation the address $|a|$ and the data vector $x = (x_0, \ldots, x_{n-1})$. Then $\text{DSA}_n(a,x) = x_{|a|}$. Let $g = \text{DSA}_n(b,x) = x_{|b|}$. Then $\pi\text{-OBDD}(f) = \pi\text{-OBDD}(g) = 2n+1$ for the variable ordering $(a_{k-1}, \ldots, a_0, b_{k-1}, \ldots, b_0, x_0, \ldots, x_{n-1})$ but $\text{OBDD}(f \oplus g) = \Theta(n^2)$ for the operation $\oplus$ (logical EXOR). This lower bound proof is quite involved. In Section 8, lower bound techniques are presented and it is shown that one-way communication complexity is the essential technique for lower bounds on the OBDD size as long as one is interested in distinguishing polynomial from exponential size. For bounds that are asymptotically tight we have to prove for our example $f \oplus g$ that $\Omega(n)$ levels have size $\Omega(n)$ or enough levels have size $\omega(n)$. The proof is difficult, since for each level there exists a variable ordering where this level has size $o(n)$.

7. More general BDD variants with good algorithmic properties

Here we present those three BDD variants which are less restricted than OBDDs and have nevertheless good algorithmic properties.

FBDDs are read-once BDDs which need not be oblivious. For read-once BDDs, the satisfiability test is a simple connectivity check. The synthesis of linear-size FBDDs (functions which even have linear OBDD size) may lead to functions with exponential FBDD size. We have to restrict FBDDs in a way similar to the restriction of OBDDs by π-OBDDs with a fixed variable ordering. Such an approach has been suggested independently by Gergov and Meinel [19] and Sieling and Wegener [34]. The result is called graph driven FBDDs, since the lists describing variable orderings are replaced with graphs describing so-called graph orderings. For each input a, we like to prescribe a variable ordering π_a. However, the variable orderings π_a have to be compatible, i.e., each boolean function f on X has to be representable by an FBDD where the computation path for the input a tests the variables in the ordering π_a (where it is allowed to skip tests). It turns out that it is necessary and sufficient that a graph ordering is described by a *complete* pseudo-FBDD, i.e., each path contains an x_i-node for each x_i, and the computation path for a contains the variables in the ordering π_a. This pseudo-FBDD G (which indeed needs no sink) is called graph ordering. All FBDDs G' that respect for each input a the variable ordering π_a contained in G are called G-FBDDs. Since one needs representations of G and G', one only works with graph orderings G of (small) polynomial size. The reader should verify that, for $n=3$, it is possible to choose $\pi_a=(x_1,x_2,x_3)$ for all a where $a_1=1$ and to choose $\pi_a=(x_1,x_3,x_2)$ otherwise. However, it is not possible to choose $\pi_a=(x_1,x_2,x_3)$ for all a where $a_1=1$ and to choose $\pi_a=(x_2,x_1,x_3)$ otherwise. Each OBDD is the π-OBDD for some π and each FBDD is the G-FBDD for some graph ordering G.

Fixing the graph ordering G, we obtain efficient algorithms for the synthesis with integrated reduction, the equality test, and the satisfiability test. However, a replacement of a single variable with a constant may cause an exponential-size blow-up for a fixed graph ordering. This can be prevented by special graph orderings if it is known in advance which variables are replaced. Another possibility which works in certain applications is to change the graph ordering. Then the replacement can be performed like in the OBDD case, see Fig. 7. The real problem is the graph-ordering problem, i.e., the problem of finding suitable graph orderings. The known heuristics do not lead to satisfactory results.

FBDDs are read-once but not necessarily oblivious BDDs. The next approach is to consider oblivious BDDs that are not necessarily read-once. Again we have to investigate s-BDDs using a fixed sequence of variables labelling the levels. Then it is easy to generalize the OBDD synthesis algorithm without reduction to a polynomial-time s-BDD synthesis algorithm, where the size of the result is bounded by the product of the sizes of the inputs. Replacements by constants without reductions can be done as for OBDDs. Bollig et al. [6] have shown that the satisfiability test is NP-complete even if $s=(s_1,\ldots,s_l)$ where $l=2n$ and $(s_1,\ldots,s_n)$ and $(s_{n+1},\ldots,s_{2n})$ are variable orderings. Hence, they have introduced k-OBDDs which are s-oblivious BDDs where $s=(s_1,\ldots,s_l)$, $l=kn$, and each block $(s_{(i-1)n+1},\ldots,s_{in})$, $1\leqslant i\leqslant k$, represents the *same* variable ordering π. Moreover, they have designed polynomial-time satisfiability and equality tests for π-k-OBDDs. The algorithms are only efficient for small k, since $2k-1$ occurs in the exponent of the run time. A further obstacle is that the

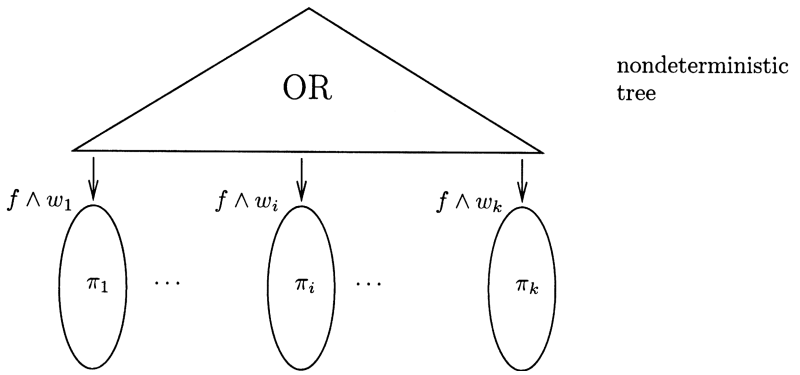


Fig. 9. A partitioned (nondeterministic) OBDD with k parts representing $f \wedge w_1, \ldots, f \wedge w_k$.

complexity of the minimization problem is unknown and only very inefficient minimization algorithms are available.

The third variant has already found many applications. Nondeterministic BDDs (OBDDs, FBDDs,...) allow unlabelled branching nodes where the computation path nondeterministically follows one of the outgoing edges. An input a is accepted if at least one legal computation path reaches the 1-sink. The problem with nondeterministic complexity classes is that they are not closed under negation (i.e., the NP$\neq$coNP-hypothesis). The negation of a nondeterministic linear-size OBDD can indeed lead to an exponential-size blow-up. This holds even if nondeterministic nodes are only allowed before the decision nodes. Narayan et al. [26] have therefore looked for a further restriction to make negation easy.

The nondeterministic decisions are made in the beginning. The number of possibilities $k = k(n)$ is fixed for each n. The k parts represent $f \wedge w_1, \ldots, f \wedge w_k$ for fixed window functions $w_1, \ldots, w_k$ describing on which parts of $\{0,1\}^n$ the function has to be represented. To represent all parts of f the condition $w_1 + \cdots + w_k = 1$ is necessary. In most cases, $w_i \wedge w_j = 0$ for $i \neq j$. Then $w_1^{-1}(1), \ldots, w_k^{-1}(1)$ are a partition of $\{0,1\}^n$ leading to the notion of *partitioned* BDDs. Each part may use its own variable ordering π_i, $1 \leqslant i \leqslant k$. The π_i-OBDD size of w_i should be small (Fig. 9).

The operation $\otimes \in \{\mathrm{OR}, \mathrm{AND}, \mathrm{EXOR}\}$ can be performed on the parts independently of each other, since

$$(f \otimes g) \wedge w_i = (f \wedge w_i) \otimes (g \wedge w_i)$$

in all these cases. However, $\overline{f \wedge w_i} \neq \bar{f} \wedge w_i$ in general. But

$$\overline{(f \wedge w_i)} \wedge w_i = (\bar{f} + \bar{w}_i) \wedge w_i = \bar{f} \wedge w_i.$$

Hence, negation can be performed as negation of each part followed by an AND-synthesis with the corresponding window function. A function f is satisfiable iff one of the functions $f \wedge w_i$ is satisfiable. Moreover,

$$f = g \leftrightarrow f \wedge w_i = g \wedge w_i \quad \text{for all } i.$$

Table 1
Algorithmic properties of BDD models

	Synthesis	Minimization reduction	Equality test	Replacement by constants
π-OBDDs	$\|G_f\| \cdot \|G_g\|$	$\|G_f\|$	$\|G_f\| + \|G_g\|$	$\|G_f\|$
G-FBDDs	$\|G_f\| \cdot \|G_g\| \cdot \|G\|$	$\|G_f\|$	$\|G_f\| + \|G_g\|$	Exponential
k-OBDDs	$\|G_f\| \cdot \|G_g\|$	?	$(\|G_f\| \cdot \|G_g\|)^{2k-1}$	$\|G_f\|$
Part. OBDDs	$\|G_f\| \cdot \|G_g\|$	$\|G_f\|$	$\|G_f\| + \|G_g\|$	Exponential

Hence, the OBDD packages can be used for manipulating partitioned OBDDs. There remains a drop of bitterness, since the replacement by constants operation can cause difficulties. Assume that $w_1 = x_1$ and $w_2 = \bar{x}_1$ are the window functions. A function f may have linear-size partitioned BDDs since $f \wedge x_1$ has a small π_1-OBDD while $f \wedge \bar{x}_1$ has a small π_2-OBDD for some $\pi_2 \neq \pi_1$. Replacing x_1 with 1, the second part has to represent $f_{|x_1=1} \wedge \bar{x}_1$ which may have exponential π_2-OBDD size. Knowing in advance the variables which will be replaced with constants one can work with partitioned BDDs whose window functions do not essentially depend on these variables.

Bollig and Wegener [8] have investigated the representational power of partitioned BDDs. Even partitioned BDDs using only one variable ordering π for all parts can lead to an exponential decrease of the representation size. Then it is possible to halve the number of parts with only a polynomial-size blow-up. Using the full power of partitioned BDDs, namely different variable orderings for the different parts, already one part less can lead to an exponential-size blow-up (even if one may choose new window functions and new variable orderings).

There are only heuristic algorithms to generate the window functions. Such algorithms and experimental results proving the practical usefulness of this approach have been presented by Jain et al. [22].

The algorithmic properties of the four BDD models which have found applications are described in the following table. Table 1 is restricted to the four most essential operations.

8. The representation size of selected functions and lower bound techniques

We are only able to present very few of the known results (see [39] for a list of known results).

OBDDs can represent several important functions in polynomial size:

- the direct storage access function DSA or multiplexer in size $2n + 1$,
- all bits of n-bit addition in size $9n-5$ (for the variable ordering $(x_{n-1}, y_{n-1}, \ldots, x_0, y_0)$ while the more "natural" ordering $(x_0, y_0, \ldots, x_{n-1}, y_{n-1})$ leads to quadratic OBDD size),
- conjunction or disjunction of n bits in size $n + 2$,

- parity or EXOR of n bits in size $2n+1$,
- arbitrary symmetric functions in size at most n^2+2,
- functions representable by read-once formulas, i.e., formulas of size $n-1$ where the n inputs are different variables in size $1.360n^{\beta}$ where $\beta=\log_4(3+\sqrt{5})<1.195$.

All results but the last one are folklore results, the last one has been obtained by Sauerhoff et al. [31]. With respect to DNFs (disjunctive normal forms), another classical representation type, only DSA, conjunction, and disjunction have polynomial size, all other considered functions have exponential DNF size. This partially explains the success of OBDDs.

However, there are two generalized storage access functions where bits may serve as address bits *and* as data bits. The hidden weighted bit function HWB_n is defined by

$$\mathrm{HWB}_n(a_1,\ldots,a_n)=a_{\|a\|},$$

where $\|a\|=a_1+\cdots+a_n$ is the number of ones in the input and $a_0:=0$. The indirect storage access function ISA_n is defined on an address vector $a=(a_{k-1},\ldots,a_0)$ and a vector $x=(x_0,\ldots,x_{n-1})$, $n=2^k$. The address vector is interpreted as the binary number with value $|a|$ pointing to a block $x(a)=(x_{|a|},\ldots,x_{|a|+k-1})$ (the indices are taking mod n) in x. Then

$$\mathrm{ISA}_n(a,x)=x_{|x(a)|}.$$

One-way communication complexity (see the end of this section) leads to a $2^{n/5}$ lower bound for HWB_n [13] and an $\Omega(2^{n/\log n})$ bound for ISA_n. These functions are interesting, since they belong to the class of "very simple functions" with exponential OBDD size. The reader can easily verify that both functions have $O(n^2)$ 2-OBDDs where the address is computed in the first part and the output bit is tested in the second part. Since this approach can be realized for ISA_n by a decision *tree* whose repeated tests can be eliminated, we also obtain an $O(n^2)$ FBDD for ISA_n. The design of an $O(n^2)$ FBDD for HWB_n is more involved [34]. The design of $O(n^3)$ partitioned BDDs with n parts (using the same variable ordering) is based on the well-known guess-and-verify mode. The address is guessed, i.e., for HWB_n we use the window function w_i checking whether the input contains exactly i ones. Then

$$(\mathrm{HWB}_n\wedge w_i)(a)=1\leftrightarrow\|a\|=i \text{ and } a_i=1.$$

What about multiplication? This is perhaps the most important function. A benchmark circuit realizing 16-bit multiplication causes already difficulties. Moreover, lower bounds for multiplication for all considered BDD variants lead to lower bounds for squaring, the computation of the n most significant bits of the inverse of an n-bit number, and division. This has been shown by special reductions namely read-once projections which allow to transfer lower bounds for all the considered BDD variants. Wegener [38] has reduced multiplication with read-once projections to the other mentioned arithmetic functions.

Bryant [13] used the fooling set method to obtain lower bounds on the communication complexity of the middle bit of multiplication leading to a $2^{n/8}$ lower bound on the OBDD size of this function. This result also shows the drawback of the asymptotic

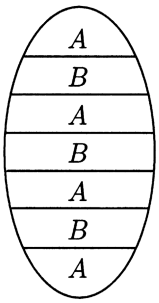


Fig. 10. A partition of an oblivious BDD into layers.

considerations. The lower bound does not exclude that even the 256-bit multiplication can be represented in reasonable size. This implies the importance of more precise lower bounds (see also the discussion in Section 6). Woelfel [40] was the first to obtain a better bound of size $2^{n/2}/192$ proving the difficulty of 64-bit multiplication. His upper bound of $(7/3)\cdot 2^{4n/3}$ proves that the middle bit of 16-bit multiplication can be represented in reasonable size.

It took quite a long time until an exponential lower bound for the FBDD size of multiplication has been proved. The lower bound due to Ponzio [29] is of size $2^{\Omega(n^{1/3})}$. Recently, Bollig and Woelfel [10] have obtained a lower bound of size $2^{(n-9)/4}$.

Exponential lower bounds for oblivious BDDs of linear length (and still nonpolynomial lower bounds for length $o(n\log n/\log\log n)$) representing multiplication have been proved by Gergov [18]. This bound includes k-OBDDs for constant k and partitioned BDDs with a small number of parts. The last result follows, since a partitioned BDD with k parts can be simulated by a (kn)-length oblivious BDD of the same size.

These lower bounds prove that the verification of multipliers and dividers is a difficult task.

We cannot present the lower bound proofs in this survey paper but we can describe the main lower bound techniques. Almost all lower bounds on oblivious BDDs are based on lower bounds in communication complexity (see the monographs of Kushilevitz and Nisan [23] and Hromkovič [21]). In the general case of oblivious BDDs of linear length, by the pigeon hole principle one obtains a subfunction on $\Omega(n)$ variables and a partition of these variables into two sets A (variables given to Alice) and B (variables given to Bob) such that the *layer depth* with respect to A and B is bounded above by a constant. That is, the set of levels can be partitioned into consecutive blocks called layers such that each layer either contains only nodes labelled by variables from A or only nodes labelled by variables from B (see Fig. 10). For k-OBDDs we obtain $2k$ layers if Alice gets the first m variables and Bob gets the remaining variables.

The communication complexity of a function f with respect to a partition of X into A and B is the minimal number of bits that have to be interchanged between Alice knowing the values of the inputs from A and Bob knowing the values of the inputs from B such that one of them knows the value of f on the actual input. Before getting

the actual parts of the inputs, Alice and Bob get an agreement on the communication protocol. If the oblivious BDD has a small number of k layers and a small size $|G|$, the communication complexity $C_{k-1}(f)$ of a $(k-1)$-round communication to compute f is bounded above by $(k-1)\lceil\log|G|\rceil$. The protocol is the following one. The player holding the variables from the first layer computes the first part of the communication path and communicates with $\lceil\log|G|\rceil$ bits the number of the first node labeled by a variable held by the other player. This player goes on in a similar way. After $k-1$ rounds of communication, the player holding the variables of the last layer has enough information to compute the label of the sink reached by the actual computation path. Hence, lower bounds on the communication complexity lead to lower bounds on the size of oblivious BDDs. For k-OBDDs it is sufficient to consider protocols restricted to $2k-1$ rounds of communication. In particular, lower bounds on the OBDD size can be obtained by lower bounds on one-round communication.

The situation for nonoblivious BDDs is totally different. In order to distinguish polynomial size from nonpolynomial or exponential size one can assume without loss of generality that for each node pair (v,w) the same set of variables has been tested on each path from v to w. The first exponential lower bounds for FBDDs have been proved by defining a cut through the FBDD realizing f and by proving that not too many inputs can pass the same cut node [37,41]. This method has been generalized by Simon and Szegedy [35] by assigning weights to the inputs. This cut-and-paste technique is quite successful for FBDDs, but it cannot be generalized to BDDs where every graph theoretical path contains for each variable x_i at most two (or even k) x_i-nodes.

A generalized approach has been started by Borodin et al. [11] and Okol'nishnikova [28]. We discuss this approach for FBDDs. A small-size FBDD has a short cut of those nodes where $\lfloor n/2\rfloor$ of the n variables have been tested. Each cut node defines a rectangle function $r(X)=r_1(X_1)\wedge r_2(X_2)$ where $|X_1|=\lfloor n/2\rfloor, |X_2|=\lceil n/2\rceil, X_1\cap X_2=\emptyset$. The function $r_1(X_1)$ computes 1 if the computation path reaches the cut node and the function $r_2(X_2)$ computes 1 if the computation path starting at the cut node reaches the 1-sink. Obviously, f is the disjunction of the rectangle functions for all cut nodes (see Fig. 11). The crucial point is that different rectangles may use different partitions of X. For a lower bound on the FBDD size it is sufficient to prove that $|r^{-1}(1)|$ is small compared to $|f^{-1}(1)|$ and this has to hold for all rectangle functions $r\leqslant f$. The combinatorial approach to prove such a property is to prove that f has the rectangle balance property, i.e., all rectangles $r=r_1(X_1)\wedge r_2(X_2)$ have the property that $r^{-1}(1)$ contains approximately as many elements from $f^{-1}(1)$ as from $f^{-1}(0)$. This implies that only small rectangles can be monochromatic, i.e., can contain only elements from $f^{-1}(1)$ or only elements from $f^{-1}(0)$.

9. On BDD-based graph algorithms

The typical applications of BDD techniques are hardware verification, model checking, and many CAD problems. We present here applications which may become important in the near future. The area of algorithmic graph problems has been investigated intensively. Efficient algorithms have been developed and many of them are at least

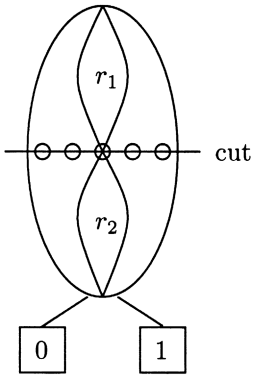


Fig. 11. An illustration of rectangle functions $r = r_1(X_1) \wedge r_2(X_2)$.

close to optimal. These algorithms assume that the graph is given explicitly, i.e., the graph is described by adjacency lists or its adjacency matrix. Moreover, the information fits into the main memory. We do not believe that, in these situations, BDD techniques will lead to improvements. However, graphs considered in applications get larger and larger.

A graph is called "large" if its explicit description does not fit into the main memory and most information is stored in the external memory. There is a new area investigating the so-called external graph algorithms.

Many graphs in applications are not random ones but have some structure allowing a more succinct representation than the explicit representation. We may encode the vertices by words from $\{0,1\}^n$ (adding perhaps some dummy vertices). The edge relation can be described as a boolean function $E : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ such that $E(x,y) = 1$ if there is an edge between vertex x and vertex y (or from x to y). The OBDD representation of E may be much more succinct even than a list of all vertices. Graph algorithms working on OBDDs may take advantage from this succinct representation.

A graph is called "very large" if it cannot be described explicitly (because of limited time and space resources). Such graphs are only described implicitly. Examples of such graphs occur, e.g., in rail traffic simulation considering all combinations of trains, stations, track sections, time slots, engineers, etc. All algorithms on such graphs have to work on implicit graph representations.

The complexity of graph problems on implicitly defined graphs may be different from the complexity of the same problem on explicitly described graphs. Feigenbaum et al. [17] have proved that the connectivity problem on graphs represented by OBDDs for the edge relation is NP-hard. This implies that we can only hope for heuristic algorithms. Nevertheless, OBDD-based algorithms may beat external graph algorithms on large graphs and implicit graph algorithms are the only choice on very large graphs.

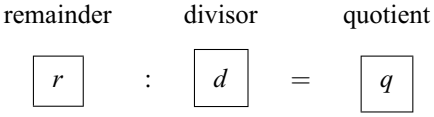
However, we have to look for a new type of algorithms. Loops like "for all $v \in V$ do sequentially ..." are intractable. OBDD synthesis works for all $v \in V$ "in parallel."

The only implemented graph algorithm which works on some very large graphs (10^{27} vertices and 10^{36} edges) is an algorithm for the maxflow problem on 0-1-networks due to Hachtel and Somenzi [20]. They have transformed the classical maxflow algorithm of Malhotra et al. [25] into an OBDD-based algorithm.

10. An OBDD-based approach to avoiding the Pentium bug

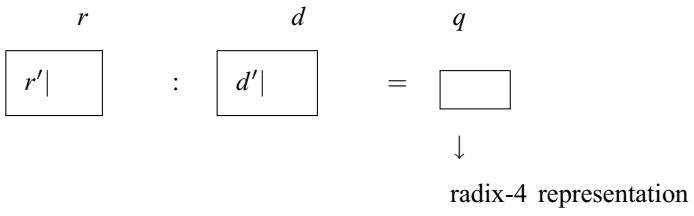
It is well known that a faulty Pentium processor was delivered in the early nineties of the last century. For certain inputs, the result of division was wrong. The exchange of all processors cost approximately 450 million dollars. Although we have seen that it is not possible to represent the division of 64-bit integers by OBDDs (because of lack of space), Bryant [14] has shown how this failure could have been avoided using OBDDs.

In order to describe this approach, we have to describe the Pentium divider. Not all details of the divider are publicly known, but the design basically follows the design of Atkins [2]. The background is the school method for division. The dividend is called remainder, since it changes during the procedure. Hence, the general situation looks as follows:



It is assumed that initially $1 \leqslant r < 2$ and $1 \leqslant d < 2$. The school method has to consider *all* bits of r and *all* bits of d to produce the first bit q_0 of q. Afterwards, the old remainder r is replaced with its new value $(r - q_0 \cdot d) \cdot 2$.

The first idea is to produce the quotient in radix-4 representation, a redundant number representation where $x = (x_0, x_{-1}, x_{-2}, \ldots)$, $x_i \in \{-3, -2, -1, 0, 1, 2, 3\}$ represents the sum of all $x_i 4^i$. This redundant representation allows the addition of two numbers in constant parallel time. The idea is that the redundancy of the representation allows the computation of a legal q_0 based on a few bits from r and d (we assume that these numbers are represented as binary numbers, the fact that r is represented by r_1 and r_2 with $r = r_1 + r_2$ and numbers r_1 and r_2 in two's complement representation is not important here).



A legal q_0 has to be computed from r' consisting of 7 bits (one sign bit, since remainders now may be negative, 3 bits to the left and 3 bits to the right of the binary point) and d' consisting of 5 bits (always the bit 1 to the left and 4 bits to the right of the binary point).

The new remainder equals

$$(r - q_0 \cdot d) \cdot 4,$$

since the "value" of each position now increases by a factor of 4. The computation of $q_0 \cdot d$ is trivial, if $q_0 \in \{-1, 0, 1\}$, it is a simple shift, if $q_0 \in \{-2, 2\}$, but it is a shift and an addition, if $q_0 \in \{-3, 3\}$. Therefore, the next idea is to allow only q_0-values from $\{-2, -1, 0, 1, 2\}$. A simple calculation shows the following facts:

- $q_0 = 2$ is legal $\Leftrightarrow 4d \leqslant 3r \leqslant 8d$,
- $q_0 = 1$ is legal $\Leftrightarrow d \leqslant 3r \leqslant 5d$,
- $q_0 = 0$ is legal $\Leftrightarrow -2d \leqslant 3r \leqslant 2d$,
- $q_0 = -1$ is legal $\Leftrightarrow -5d \leqslant 3r \leqslant -d$,
- $q_0 = -2$ is legal $\Leftrightarrow -8d \leqslant 3r \leqslant -4d$.

Since these intervals overlap, we may still hope that a legal q_0 can be computed from the prefixes r' and d'. It has to be ensured that the invariant $-8d \leqslant 3r \leqslant 8d$ is always fulfilled. The divider gets a 128×16-table containing for each possible r', d' a legal q_0-entry. However, some entries of this table were wrong. One may ask, why these wrong entries have not been found during the testing process. The wrong entries were at positions where $3r \approx 8d$. Since in the beginning $1 \leqslant r < 2$ and $1 \leqslant d < 2$, the first entries which are looked up in the table are far away from the wrong entries. There are only few numbers, where the wrong entries are looked up.

What is the verification task? Let $q_0(r', d')$ be the table entry at position (r', d'). Then we have to verify the following statement where R contains the interval of allowed r-values:

$$\forall d, 1 \leqslant d < 2, \ \forall r \in R:$$

$$-8d \leqslant 3r \leqslant 8d \Rightarrow (r - q_0(r', d') \cdot d) \cdot 4 \in R \quad \text{and}$$

$$-8d \leqslant 3 \cdot (r - q_0(r', d') \cdot d) \leqslant 8d,$$

where r' and d' are the described prefixes of r and d, resp. The formula can be described by a circuit using a conservative design style (in order to produce no failures here). In this circuit we include a circuit realizing the table entries $(r', d') \rightarrow \{-2, -1, 0, +1, +2\}$. This is possible, since we only have 11 variables. Now we can apply OBDD techniques. Using the synthesis algorithm the circuit is transformed into an OBDD. Afterwards, the quantification process is applied to the d- and r-variables. If we obtain the constant 1, the table is verified. It is even possible to construct a table by OBDD techniques. Then the table entries are described by 7 r'-variables, 4 d'-variables, and 3 variables describing $q_0(r', d')$. The satisfying inputs (if existent) of the resulting OBDD describe the legal table entries.

11. Conclusion

We have described many aspects of the theory of BDDs and have discussed how BDD techniques can be applied to real-world problems. One can expect that BDDs

will stay a major computation model in complexity theory. However, the still open problems are hard, in particular, exponential-size lower bounds for more general BDD models. In applications, BDDs will become a standard tool in many areas and one can hope that practitioners do not apply only OBDDs but also more sophisticated models.

References

[1] M. Ajtai, A non-linear time bound for boolean branching programs, FOCS'99, IEEE Computer Society Press, Los Alamitos, Calif., 1999, pp. 60–70.
[2] D.F. Atkins, Higher-radix division using estimates of divisor and remainder, IEEE Trans. Comput. 17 (1968) 925–934.
[3] P. Beame, T.S. Jayram, M. Saks, Time-space tradeoffs for branching programs. Journal of Computer and System Sciences 63 (2001) 542–572.
[4] P. Beame, M. Saks, X. Sun, E. Vee, Super-linear time-space tradeoff lower bounds for randomized computation, FOCS'2000, IEEE Computer Society Press, Los Alamitos, Calif., 2000, pp. 169–179.
[5] B. Bollig, M. Löbbing, I. Wegener, On the effect of local changes in the variable ordering of ordered decision diagrams, Inform. Process. Lett. 59 (1996) 233–239.
[6] B. Bollig, M. Sauerhoff, D. Sieling, I. Wegener, Hierarchy theorems for k-OBDDs and k-IBDDs, Theoret. Comput. Sci. 205 (1998) 45–60.
[7] B. Bollig, I. Wegener, Improving the variable ordering of OBDDs is *NP*-complete, IEEE Trans. Comput. 45 (1996) 993–1002.
[8] B. Bollig, I. Wegener, Complexity theoretical results on partitioned (nondeterministic) binary decision diagrams, Theory Comput. Systems 32 (1999) 487–503.
[9] B. Bollig, I. Wegener, Asymptotically optimal lower bounds for OBDDs and the solution of some basic OBDD problems, ICALP'2000, Lecture Notes in Computer Science, Vol. 1853, Springer, Berlin, 2000, pp. 187–198.
[10] B. Bollig, P. Woelfel, A read-once branching program lower bound of $\Omega(2^{n/4})$ for integer multiplication using universal hashing, STOC'2001, ACM, New York, 2001, pp. 419–424.
[11] A. Borodin, A. Razborov, P. Smolensky, On lower bounds for read-k-times branching programs, Comput. Complexity 3 (1993) 1–18.
[12] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Trans. Comput. 35 (1986) 677–691.
[13] R.E. Bryant, On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication, IEEE Trans. Comput. 40 (1991) 205–213.
[14] R.E. Bryant, Bit-level analysis of an SRT divider, ACM and IEEE Computer Society, Design Automat. Conf. 33 (1996) 661–665.
[15] A. Cobham, The recognition problem for the set of perfect squares, Seventh IEEE Symposium on Switching and Automata Theory, IEEE Computer Society Press, Los Alamitos, Calif., 1966, pp. 78–87.
[16] R. Drechsler, N. Göckel, Minimization of BDDs by evolutionary algorithms, IWLS'97, International Workshop on Logic Synthesis, 1997, Workshop handouts.
[17] J. Feigenbaum, S. Kannan, M.Y. Vardi, M. Viswanathan, Complexity of graphs represented as OBDDs, STACS'98, Lecture Notes in Computer Science, Vol. 1373, Springer, Berlin, 1998, pp. 216–226.
[18] J. Gergov, Time-space tradeoffs for integer multiplication on various types of input oblivious sequential machines, Inform. Process. Lett. 51 (1994) 265–269.
[19] J. Gergov, C. Meinel, Efficient Boolean manipulation with OBDD's can be extended to FBDD's, IEEE Trans. Comput. 43 (1994) 1197–1209.
[20] G.D. Hachtel, F. Somenzi, A symbolic algorithm for maximum flow in 0–1 networks, Formal Methods System Design 10 (1997) 207–219.
[21] J. Hromkovič, Communication Complexity and Parallel Computing, Springer, Berlin, 1997.
[22] J. Jain, K. Mohanram, D. Moundanos, I. Wegener, Y. Lu, Analysis of composition, and how to obtain smaller canonical graphs, ACM, New York, Design Automat. Conf. 37 (2000) 681–686.
[23] E. Kushilevitz, N. Nisan, Communication Complexity, Cambridge University Press, Cambridge, 1997.

[24] C.Y. Lee, Representation of switching circuits by binary-decision programs, Bell Systems Technical J. 38 (1959) 985–999.
[25] V.M. Malhotra, M. Pramodh Kumar, S.N. Maheshwary, An $O(|V|^3)$ algorithm for finding maximum flows in networks, Inform. Process. Lett. 7 (1978) 277–278.
[26] A. Narayan, J. Jain, M. Fujita, A. Sangiovanni-Vincentelli, Partitioned ROBDDs—a compact, canonical and efficiently manipulable representation for Boolean functions, ICCAD'96, ACM and IEEE Computer Society, 1996, pp. 547–554.
[27] E.I. Nechiporuk, A Boolean function, Sov. Math. Dokl. 7 (1996) 999–1000.
[28] E.A. Okol'nishikova, On lower bounds for branching programs, Siberian Adv. Math. 3 (1993) 157–166.
[29] S. Ponzio, A lower bound for integer multiplication with read-once branching programs, SIAM J. Comput. 28 (1998) 798–815.
[30] R. Rudell, Dynamic variable ordering for ordered binary decision diagrams, ICCAD'93, ACM and IEEE Computer Society, 1993, pp. 42–47.
[31] M. Sauerhoff, I. Wegener, R. Werchner, Optimal ordered binary decision diagrams for read-once formulas, Discrete Appl. Math. 103 (2000) 237–258.
[32] D. Sieling, On the existence of polynomial time approximation schemes for OBDD minimization, STACS'98, Lecture Notes in Computer Science, Vol. 1373, Springer, Berlin, 1998, pp. 205–215.
[33] D. Sieling, I. Wegener, Reduction of OBDDs in linear time, Inform. Process. Lett. 48 (1993) 139–144.
[34] D. Sieling, I. Wegener, Graph driven BDDs—a new data structure for Boolean functions, Theoret. Comput. Sci. 141 (1995) 283–310.
[35] J. Simon, M. Szegedy, A new lower bound theorem for read-only-once branching programs and its applications, DIMACS Ser. Discrete Math. Theoret. Comput. Sci. 13 (1993) 183–193.
[36] F. Somenzi, CUDD. CU decision diagram package release 2.3.0, University of Colorado, Boulder, CO, 1998.
[37] I. Wegener, On the complexity of branching programs and decision trees for clique functions, J. ACM 35 (1998) 461–471.
[38] I. Wegener, Optimal lower bounds on the depth of polynomial-size threshold circuits for some arithmetic functions, Inform. Process. Lett. 46 (1993) 85–87.
[39] I. Wegener, Branching Programs and Binary Decision Diagrams—Theory and Applications, SIAM Monographs on Discrete Mathematics and Applications, 2000.
[40] P. Woelfel, New bounds on the OBDD size of integer multiplication via universal hashing, 18, STACS, Lecture Notes in Computer Science, Vol. 2010, Springer, Berlin, 2001, pp. 563–574.
[41] S. Žák, An exponential lower bound for one-time-only branching programs, MFCS'84, Lecture Notes in Computer Science, Vol. 176, Springer, Berlin, 1984, pp. 562–566.